T-Cube:
A Data Structure for Fast Extraction of Time Series from Large Datasets

Maheshkumar Sabhnani, Andrew W. Moore, Artur W. Dubrawski

April 2007
CMU-ML-07-114

ML
MACHINE LEARNING
DEPARTMENT

Carnegie Mellon

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **APR 2007**	2. REPORT TYPE	3. DATES COVERED **00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE
T-Cube: A Data Structure for Fast Extraction of Time Series from Large Datasets

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
School of Computer Science, Carnegie Mellon University,Pittsburgh,PA,15213-3891

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report introduces a data structure called T-Cube designed to dramatically improve response time to ad-hoc time series queries against large datasets. We have tested T-Cube on both synthetic and real world data (emergency room patient visits, pharmacy sales) containing millions of records. The results indicate that T-Cube responds to complex queries 1,000 times faster when compared to the state-of-the-art commercial time series extraction tools. This speedup has two main benefits: (1) It enables massive scale statistical mining of large collections of time series data, and (2) It allows its users to perform many complex ad-hoc queries without inconvenient delays. These benefits have been already found useful in applications related to practice of monitoring safety of food and agriculture, in detection of emerging patterns of failures in maintenance and supply management systems, as well as in the original application domain: bio-surveillance.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**21**	

T-Cube:
A Data Structure for Fast Extraction of Time Series from Large Datasets

Maheshkumar Sabhnani, Andrew W. Moore, Artur W. Dubrawski

April 2007
CMU-ML-07-114

The Auton Lab
Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report introduces a data structure called T-Cube designed to dramatically improve response time to ad-hoc time series queries against large datasets. We have tested T-Cube on both synthetic and real world data (emergency room patient visits, pharmacy sales) containing millions of records. The results indicate that T-Cube responds to complex queries 1,000 times faster when compared to the state-of-the-art commercial time series extraction tools. This speedup has two main benefits: (1) It enables massive scale statistical mining of large collections of time series data, and (2) It allows its users to perform many complex ad-hoc queries without inconvenient delays. These benefits have been already found useful in applications related to practice of monitoring safety of food and agriculture, in detection of emerging patterns of failures in maintenance and supply management systems, as well as in the original application domain: bio-surveillance.

Keywords: time series, databases, data structures, cached sufficient statistics

1 Introduction

Time series data is abundant in many domains including finance, weather forecasting, epidemiology, and many others. Large scale bio-surveillance programs monitor status of public health against adverse events such as outbreaks of infectious diseases and emerging patterns in public health. They rely on data collected throughout a health management system (hospital records, health insurance companies records, lab test requests and results, issued and filled prescriptions, ambulance and emergency phone service calls, etc) as well as outside of it (school/workplace absenteeism, sales of non-prescription medicines, etc). The key objective is to as early as possible and as reliably as possible detect such changes in statistics of the data sources which may be indicative of a developing public health problem. One of the challenges the users of such systems face is that of data overload. The actual number of e.g. daily transactions of drug sales in pharmacies across a sizable country may be very large. The users need tools to enable timely analysis of those massive data sources. The analyses can be performed automatically (using data mining software), however patterns discovered that way are almost always subject to a careful scrutiny through a manual drill-down. In both scenarios, massive screening of very large collections of data must be executed really fast in order to make these bio-surveillance systems useful in practice. A saving of just a few hours in detection time of an outbreak of a lethal infectious disease can yield enormous monetary and social benefits [3], [10].

Most of the kinds of data mentioned above can be interpreted as time series of interval (e.g. daily) counts of events (such as number of certain type of drugs, e.g. anti-diarrheals sold; number of patients reporting to emergency department with specific symptoms, etc). These time series can be sliced-and-diced across multiple symbolic dimensions such as location, gender and age group of patients, and so on. Computational efficiency of mining operations which one may want to apply to such data, as well as the efficiency of accessing interesting information in a manual drill-down mode, heavily depend on the efficiency of extraction of series of counts aggregated for specific values of these categorical dimensions.

This report introduces a new data structure designed to dramatically decrease the time of retrieval of such aggregates for any complex query which combines values of categorical variables. It achieves its efficiency by pre-computing and caching responses to all possible queries against the underlying temporal database of counts annotated with sets of symbolic labels, while keeping memory requirements incheck.

1.1 Importance of Ad-hoc Queries against Temporal Databases

A record of a typical transactional database contains multiple attribute-value pairs. In temporal databases, one of the key features is *date* which allows ordering entries by time and creating time series representations of the contents of the database. Other fields may be categorical (symbolic) or real-valued, or have a special type. We will focus on temporal databases with *symbolic* attributes used to characterize *demographics* of the individual entries (note that the term “demographics” is used here in its most general sense). The databases of our specific interest can be thought of as records of transactions – there is also a *count* component in them. Let us assume that all the attributes of the transactional dataset are symbolic except of the *date* field. Distinct values of such *demographic attributes* are *demographic values* or *properties*. E.g. postal code (called “zip code” in the US) is a demographic attribute and “15213” is one of possible values it can assume. Also let us define a *set of demographic properties (DPS)* to be a set of assignments of demographic values, one per each demographic attribute in the data. E.g. in a dataset containing *zip code* and *gender*, the particular combination of its values such as {15213, Male} is a *DPS*. Most databases we encounter in practice contain between ten and hundred thousand unique *DPSes*.

Often, time series queries extract data corresponding to conjunctions of demographic attribute-value pairs. A *simple query* is restricted to have only one value per demographic attribute whereas in a *complex query* each attribute can take multiple values. We show below examples of a simple and a complex query against a dataset “ed_table” which contains demographic attributes such as zip code, symptom and age group. Note that the second example query covering Pittsburgh region can contain a couple hundred zip codes. Each simple or complex query may involve only a subset of all available demographic attributes (alternatively, the attributes not represented in such query may be plugged in with the complete list of their values: it would be equivalent to “*do not care*” symbol put next to that attribute name in the select statement).

- *All senior patients having respiratory complaints* (example of a simple query):
  ```
  SELECT date, count(*) FROM ed_table
  WHERE age_group in (senior) AND syndrome in (respiratory)
  GROUP BY date;
  ```

- *All child patients in Pittsburgh region with fever or headache* (a complex query):
  ```
  SELECT date, count(*) FROM data
  WHERE zip code IN (15213, 15232,… 15237, …, 15215) AND symptom IN
  (fever, headache) AND age_group IN (child)
  GROUP BY date;
  ```

Even for databases with all demographic attributes indexed separately, simple queries require aggregation at run time which could become expensive. The number of just simple queries is exponential in the number of demographic attributes and their arities. The number of possible complex queries is very much higher than that as the exponential factor is doubled. It is thus not practical to pre-compute and store the answers to all possible queries in a straightforward fashion. Database management (DBM) systems often provide tools to pre-cook materialized views and use stored procedures which help to quickly answer a (usually small and predefined) subset of all possible complex queries. Those DBM techniques fail in the ad-hoc scenarios, where the user queries are not known in advance.

We are aware of at least two good examples of practical settings in which ad-hoc querying capability is needed. Firstly, take a database used by public health officials for bio-surveillance. These officials perform disease outbreak monitoring on a daily basis. It often involves investigation of alerts of possible problems flagged by the automated outbreak detection systems. For such investigations, the health officials need to execute large numbers of complex ad-hoc queries in order to extract data needed for interpretation of the alerts and/or for isolating their likely causes, while differentiating real outbreaks from false alarms. Such investigations need to be executed in a timely fashion, and long waits for data extracts are not acceptable. Secondly, automated statistical analyses or data mining procedures executed against such databases require access to a large number of different projections of data, if they are to comprehensively and purposively screen for possible indications of public health problems. Long waits for the corresponding extracts may (and often do) render such systems impractical as both of the example usage cases heavily rely on quick responses to complex queries. The users would dramatically benefit if caching responses to all possible queries in advance was available.

Still, data cubes typically require in order of one second or more to respond to each complex query. This latency is a substantial inconvenience to the users who want to execute multiple queries in an online fashion; data cubes are far too slow for statistical analyses requiring execution of millions of complex queries, which would take days of processing time.

2 Related Work

Standard approach to handling ad-hoc queries in commercial databases is to use On-Line Analytical Processing (OLAP). The idea relies on *data cubes*: cached data structures extracted from (usually only parts of) the original data and constructed in a form allowing for fast ad-hoc querying of pre-selected subsets of aggregated data [1]. For the sake of brevity we do not review the details of OLAP technology here, but these methods are known to suffer from long build times (typically hours for the databases of sizes and complexities similar to those used to illustrate results in this report) and large memory requirements (causing the need to rely on high-end database servers). Additionally, as we have observed empirically, data cubes still typically require in the order of one second or longer for responding to a complex query on the datasets which we tested. Such latency is a substantial inconvenience to the users who want to execute multiple queries in an online fashion. It also hampers statistical analyses which may require millions of complex queries. That could take days of processing time using industry-standard OLAP data cubes. More details and illustrative examples can be found in [3].

There are different types of implementations of data cubes based on how counts are stored internally. Relational OLAP (ROLAP) stores all counts in the data cube in the form of a relational table. Multi-dimensional OLAP (MOLAP) directly stores the counts as a multi-dimensional array (hence the response to any simple query can be obtained in constant time which makes them faster than ROLAP counterparts, but their memory requirements grow exponentially with the number of dimensions in data). A few other popular variants of data cube implementations include: Hybrid-OLAP (HOLAP) and Hierarchical OLAP. HOLAP combines the benefits of ROLAP (reasonable memory requirements) and MOLAP (faster response time). Idea of HOLAP follows intuition that the data cubes built at higher abstraction levels are denser than those build at the lower levels, and they often use MOLAP at coarser levels and ROLAP at finer levels. Hierarchical OLAP is used when the various values of a demographic attribute in data are connected to each other using some hierarchy. For e.g. date attribute can be split into year, month, or day and then aggregations at year or month level could be obtained using data cubes. Irrespective of the implementation, the goal of data cubes is to respond to simple and complex queries against large databases as fast as possible. With growing demand for data mining and statistical analyses against large databases, innovation work has been focusing on improving data cube performance [7],[8],[9].

Data cubes are closely related to another technology which originated from computer science research: *Cached Sufficient Statistics*. Similarly to data cubes, cached statistics structures pre-compute answers to queries, but they are designed cover all possible future queries (not just pre-selected subsets), and they aim at efficiency of not only data retrieval, but also their (most often in-memory) representations. AD-Trees are very good examples of such data structures.

AD-Tree [2] (All-Dimensional Tree) is designed to efficiently represent counts of all possible co-occurrences among multiple attributes of symbolic data. This is very important in many scenarios involving statistical modeling of such data, where most operations require computing aggregate counts, ratios of counts or their products. Quick access to counts of arbitrary subsets of demographic properties is essential for overall performance of analytic tools which rely on them. AD-Trees have been shown to dramatically speed-up notoriously expensive machine learning algorithms including Bayesian Network learning [2], Empirical Bayesian Screening, Decision Tree learning and Association Rule learning [5]. The attainable speedups range from one to four orders of magnitude with respect to previously known efficient implementations. These efficiencies are attainable at moderate memory requirements, which are easy to control. Moore describes in [2] the details of the structure, its construction algorithm as well as fundamental characteristics of the AD-Trees. There are also dynamic implementations of AD-Trees [6] which help grow the structure on demand and which can be more memory efficient than fundamental implementations. AD-Trees are the best of the existing solutions to symbolic data representation when it comes to very quickly responding to ad-hoc queries against large datasets.

3 T-Cube

T-Cube (Time series Cube) is an in-memory data structure designed for very fast retrieval and analysis of additive data streams such as e.g. time series of counts. It is a derivative of the idea of AD-trees. We show how the AD-Trees can be easily and efficiently extended to the domain of time series analysis. T-Cube consists of two main components: D-Cache and AD-Tree (note that because AD-Tree is designed to work with symbolic data, the T-Cube is applicable to temporal datasets with symbolic demographic attributes).

D-Cache is represented as a two-dimensional matrix with rows containing one *DPS* (a set containing one value per each demographic feature) and columns corresponding to the subsequent values of the time variable. Table 1 below shows an example structure of a D-Cache built for a retail transaction dataset whose entries have two demographic attributes (*color* and *size* of a t-shirt) and which covers '*T*' days of sales record. The result of any time series query (simple or complex) against such dataset is an aggregated time series over the rows (or *DPSes*) of the D-Cache that match the query. Hence, using D-Cache, the query response time is linear in the number of distinct *DPSes* in the data. It can be slow if the number of rows in the D-cache is large.

Table 1. D-Cache data structure for the example retail database.

	Day_1	Day_2	...	Day_T
(Red, Small)	10	15	...	20
(Red, Large)	5	1	...	7
...	...	...	...	...
(Blue, Medium)	7	5	...	9
(Green, Small)	5	4	...	10

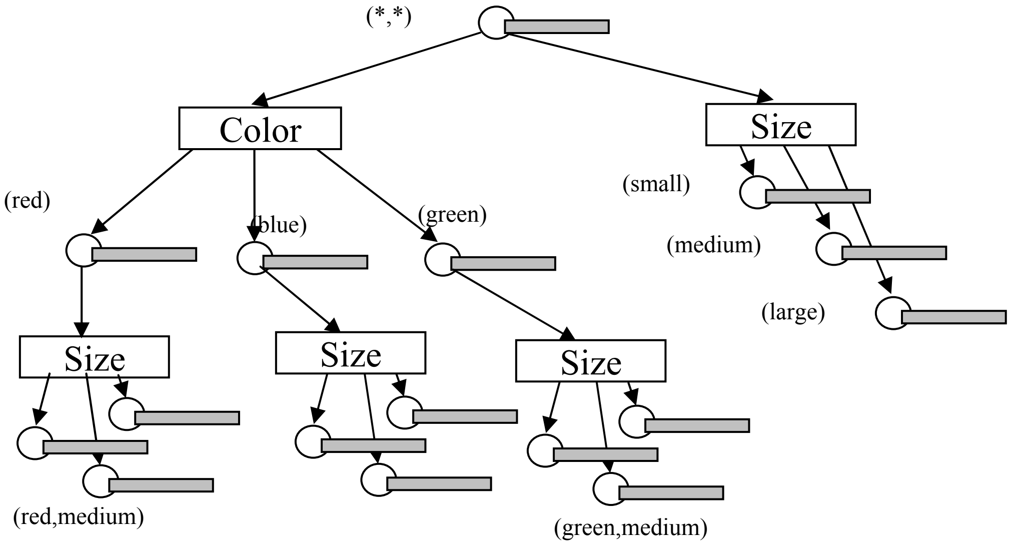

Figure 1. Fully-developed AD-Tree for the example retail database.

The goal of using the AD-Tree structure is to reduce the query response time from linear to logarithmic in the number of rows in D-Cache. We build the AD-Tree using the rows in D-cache that contain the different *DPSes* present in the data. Usually, each data node of the AD-Tree [2] contains a single count of occurrences corresponding to the particular DPS. In T-Cube representation, the data node consists of a series of *T* counts. This time series is the result of summation of corresponding counts across all the rows in D-Cache that match the current AD-Tree data node. These matching rows are called a *leaf-list* for the corresponding data node. As we go from the top to the bottom of the tree, the sizes of leaf-lists decrease and in fully developed trees the leaf nodes have leaf-lists of size one pointing to a single row in the D-Cache. Therefore such AD-Tree combined with the D-Cache store responses to all possible atomic time series queries against the underlying database. Figure 1 depicts the fully developed AD-Tree representation of the t-shirt database shown in Table 1. The data nodes with time series are shown as circles with shaded bars and the vary nodes over demographic attributes are depicted with rectangles. Note that answering more general as well as complex time series queries can be accomplished very quickly by navigating the AD-Tree and performing simple arithmetic operations on the vectors of counts pointed to by the traversed data nodes. The attainable efficiencies are analogous to those reported in the fundamental Ad-Tree paper [2].

4 Empirical Evaluation

4.1 Data

We tested T-Cube on three different datasets: two real-world and one synthetic. All the datasets used in the experiments consisted of records collected over the period of one year, with counts aggregated daily. The datasets vary in the number of records (volume), number of demographic attributes (dimensionality) and the number of distinct values of each attribute (arity). Even though the first two datasets are related to the domain of bio-surveillance, their size and characteristics are similar to data that can be found in other domains: a few attributes of high arity (zip codes, business names, etc.) and many attributes of low arity (gender, company sizes, etc.).

4.1.1 Chief Complaint Data (ED)

Emergency room chief complaint (ED) dataset contains hospital emergency room patient visit records from four US states (PA, NJ, OH and UT). Each record consists of the following attributes: visit *Date*, *Syndrome*, patient's home *Zip Code*, *Age Group*, *Gender* and *Count*. The patient's home *Zip Code* has 21,179 distinct values. Note that even though the involved hospitals are located in one of the four states listed above, the patients come from all over the country. Attribute *Syndrome* represents the chief complaint reported by the patient and has 8 distinct values (such as: Respiratory, Gastrointesitnal, Constitutional, etc.). Attribute *Age Group* could take one of the three values: Adult, Child and Unknown. Similarly attribute *Gender* could also take three values: Male, Female, and Unknown. The dataset had approximately 3.4 million records and it does not contain personal information of the patients as well as any means of identifying the involved individuals. There is a total of 120,604 *DPSes* in the *ED* data.

4.1.2 Over-The-Counter Data (OTC)

Over-the-counter (OTC) data contains the volumes of daily medication sales collected at more than 10,000 pharmacies throughout the U.S. The data has been geographically aggregated to the level of a zip code in order to preserve the privacy of the individual store operations. Each record contains the following attributes: purchase *Date*, store *Zip Code*, medicine *Category*, sale *Promotion*, and *Count*. This data covers 8,119 distinct *Zip Codes*. *Category* represents the class of medicine (e.g. cough/cold remedies, baby/child electrolytes, etc.) and it has 23 different values. Binary information about the occurrence of store promotions on medications is provided in the attribute *Promotion* as 'Y' or 'N'. Attribute *Count* represents the quantity sold. The dataset has 356,545 *DPSes* in it.

4.1.3 Binary Synthetic Data (SYN)

The real datasets encountered in our research have only a few demographic attributes (ED has four and OTC has three). In order to evaluate the utility of T-Cube in a more general setting, we have created a sparse binary synthetic data (SYN). It has a large number of attributes: 32. They include: *Date*, *Count*, *Zip Code*, and 29 *binary* demographic attributes. Each of the 29 binary attributes were 95% sparse, i.e. they took the value of '0' 95% of the time in all the records. Each record has the *Zip Code* randomly assigned from the pool of 10,000 values. Attribute *Date* spanned one year. Attribute *Count* took values randomly selected from the range of 5 to 10. 12 million records have been generated in such a way, resulting in 4.5 million *DPSes* in the data.

4.2 Results

This section reviews the experiments performed to empirically evaluate performance of T-Cube. First, we discuss the T-Cube building time and memory utilization. We then present ways of controlling exponential memory requirements of the underlying AD-Tree structure: attribute ordering, controlling tree depth, and the use of efficiencies stemming from a special treatment of the most-common-values of demographic attributes. The effects on performance have been measured with regard to simple queries and using a system with AMD Opteron 242 Dual processor CPU (1,600 MHz) and 16GB of memory. The system was running on CentOS 4 x86_64 operating system.

4.2.1 Build Time and Memory Utilization

Figure 2 shows the D-Cache and AD-Tree build times for all three datasets. The build time for D-Cache ranges between approximately 3 and 15 minutes whereas it only takes a few seconds to build the corresponding fully developed AD-Trees. It was not possible to build an AD-Tree for SYN dataset because it required a large amount of memory which exceeded the capacity of the test-bed system. These results indicate that most of the time needed to build a T-Cube is spent on building the D-Cache, and most of that time is spent accessing the data from the disk. Once all the *DPS* time series derived from data are stored in the D-Cache, construction of the AD-Tree is fast. Since the D-Cache structure is build only once for a given data set, we only pay the price once, upfront.

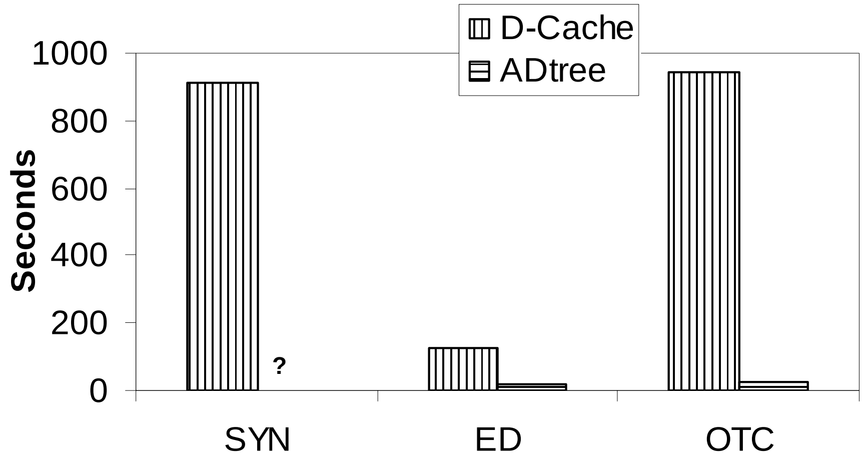

Figure 2. Build time of D-Cache and AD-Tree.

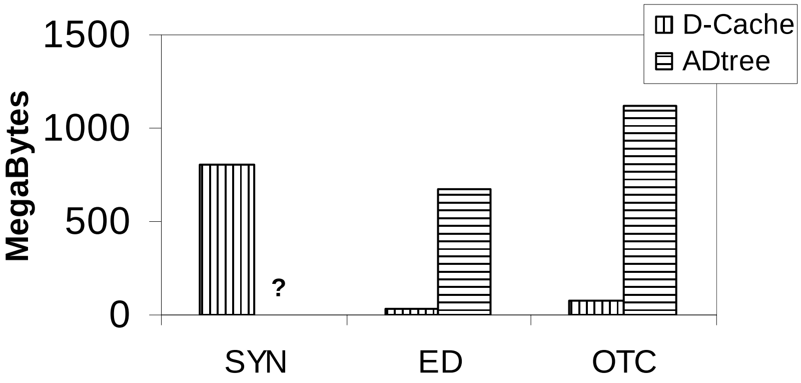

Figure 3. Memory utilization by D-Cache and AD-Tree.

Table 2. Summary of building time and memory utilization (with use of fully developed AD-Trees).

	Building Time (secs)		Memory (MB)	
	D-Cache	AD-Tree	D-Cache	AD-Tree
ED	124	16	36	676
OTC	944	26	77	1116
SYN	913	-	808	-

Figure 3 depicts memory utilization of D-Cache and AD-Tree for all three datasets. Here, the AD-Trees are much more resource hungery than the D-Caches. For instance to represent the ED dataset, we need only 36MB of memory to store D-Cache but the fully developed AD-Tree requires 676MB space (approximately 20 times more). This is due to the fact that the memory requirement of AD-Tree is exponential in the number *DPSs* present in the data. Again, for SYN dataset, AD-Tree required more than 16GB of memory and hence the corresponding result was not available. Note that the demand for memory is proportional to the length of the represented time series with respect to both D-Caches and AD-Trees.

Table 2 summarizes the memory consumption and build time results obtained using basic implementation of AD-Trees. The following sections review ways of reducing demand for memory at the cost of longer query response times.

4.2.2 Ordering of Demographic Attributes

The AD-Tree structure is inherently unbalanced: its left part is deeper than the right one. We have found empirically that expanding the lack of balance even further by arranging the demographic attributes in decreasing order of their arity leads to AD-Trees with lower numbers of nodes in total. This heuristic will make the attribute of the highest arity the root node of the tree, and the attributes of the lowest arities will end up deep in the tree. Intuitively, it would help conserve memory by preventing the attributes with high arities from being represented multiple times as children in the deeper parts of the structure.

According to this heuristic, we would arrange the ED dataset demographic attributes in the following order: patient home *Zip Code*, *Syndrome*, *Age Group*, and *Gender*.

Table 3 compares the memory requirements resulting in use of the above described heuristic vs. its direct opposite. It is clear that arranging attributes in decreasing order of their arities saves memory as compared to arranging them in the increasing order of arities, and there is no discernable effect on the query response time. We could save nearly 200MB of memory to represent the ED data while using 140 thousand less AD-Tree nodes. However, the savings were not sufficient to allow for representing the SYN dataset within the 16GB of memory.

Table 3. Effects of ordering demographic attributes.

	Memory (MB)		#nodes	
	Increasing	Decreasing	Increasing	Decreasing
ED	875	676	484k	346k
OTC	1049	1039	565k	552k
SYN	-	-	-	-

4.2.3 Controlling Depth of the Tree

The root node of the AD-Tree represents the aggregate time series that matches all combinations of the demographic properties, i.e. the leaf list of the root node is as large as the number of rows in the D-Cache. A complete tree keeps growing until the leaf nodes represent exactly one *DPS* each, at which point such nodes directly point to the individual and distinct D-Cache rows. The data node leaves in the fully grown AD-Tree will have leaf list size equal to 1.

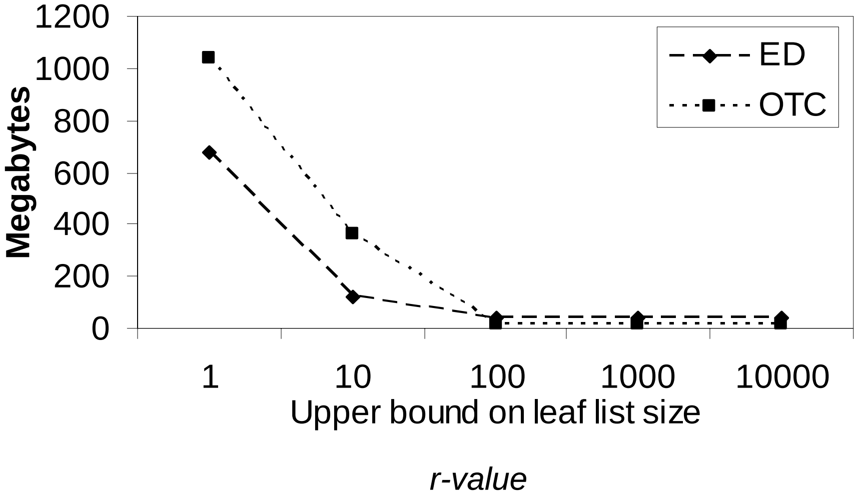

Figure 4. Memory utilization at different *r-values*.

To reduce the memory requirement, we can set a lower bound on the leaf list size of all data nodes in the tree, call it *r-value*. A data node is further expanded only if its leaf list size is greater than *r-value*. Larger *r-values* will correspond to smaller trees and hence lower memory usage. Note that with *r-value* > 1, the AD-Tree leaves will point to multiple rows in the D-Cache. Hence, queries that invoke more specific time series will have to sequentially scan the D-Cache rows which can be done in linear time, and it will reduce the expected speed of query response. Figures 4 and 5 depict memory requirements and query response times for different *r-values*.

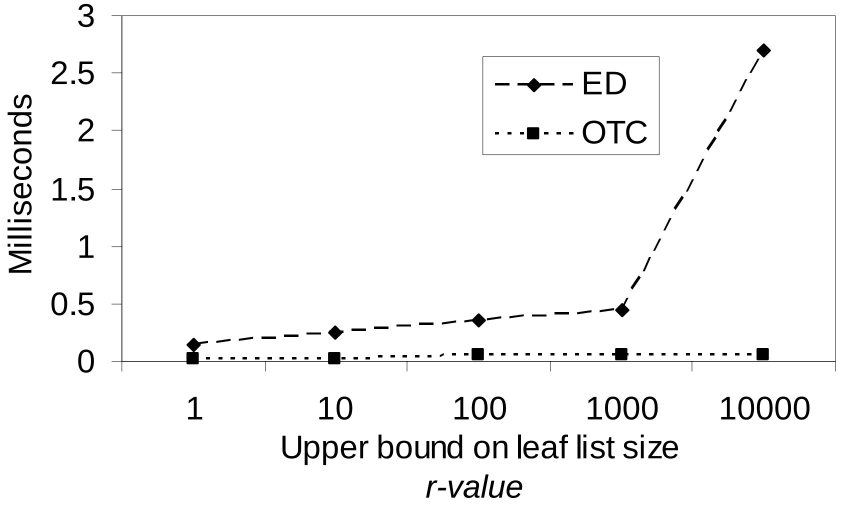

Figure 5. Average (simple) query response times for different *r-values*.

For ED data, the memory requirement falls from 676MB to 38MB (almost 20 times) when the *r-value* is changed from 1 to 100 (Figure 4). Similar effect can be seen for OTC data. Note that the figures do not show results for SYN dataset as it still needed more that 16BG of memory even when using *r-value* = 10,000.

Table 4. Sample dataset for Figure 6.

Date	Gender	Place	Count
2006/01/01	M	100	4
2006/01/01	M	300	3
2006/01/01	F	300	1
2006/01/01	M	200	3
2006/01/01	F	400	2
2006/01/02	M	200	1
2006/01/02	F	400	4
2006/01/02	M	300	2
2006/01/02	F	300	5
2006/01/02	M	200	6
2006/01/03	M	200	2
2006/01/03	F	300	1
2006/01/03	M	100	4
2006/01/03	F	300	2
2006/01/03	F	400	3

Figure 5 shows the T-Cube query response time to simple queries as a function of the tree size. Each presented result is the average response time over 1,000 randomly generated simple queries. Note that changing *r-value* from 1 to 100 drastically reduces memory requirements at a marginal difference in response time.

4.2.4 Exploiting Most Common Values

It is possible to further reduce memory requirements by exploiting redundancies in the AD-Tree structure. Moore and Lee [2] have shown that it is possible to remove one *vary* node together with the corresponding sub-tree from under each dimension nodes, and still be able to recover all the counts which were represented in the complete tree. Typically, the greatest possible savings in terms of the number of nodes in the tree can be attained if the sub-trees removed correspond to the most common value of the attribute under consideration.

(300) = (,*) – (100) – (200) – (400)*
(M, 300) = (300) – (F, 300)
(F, 400) = (F,) – (F, 300)*

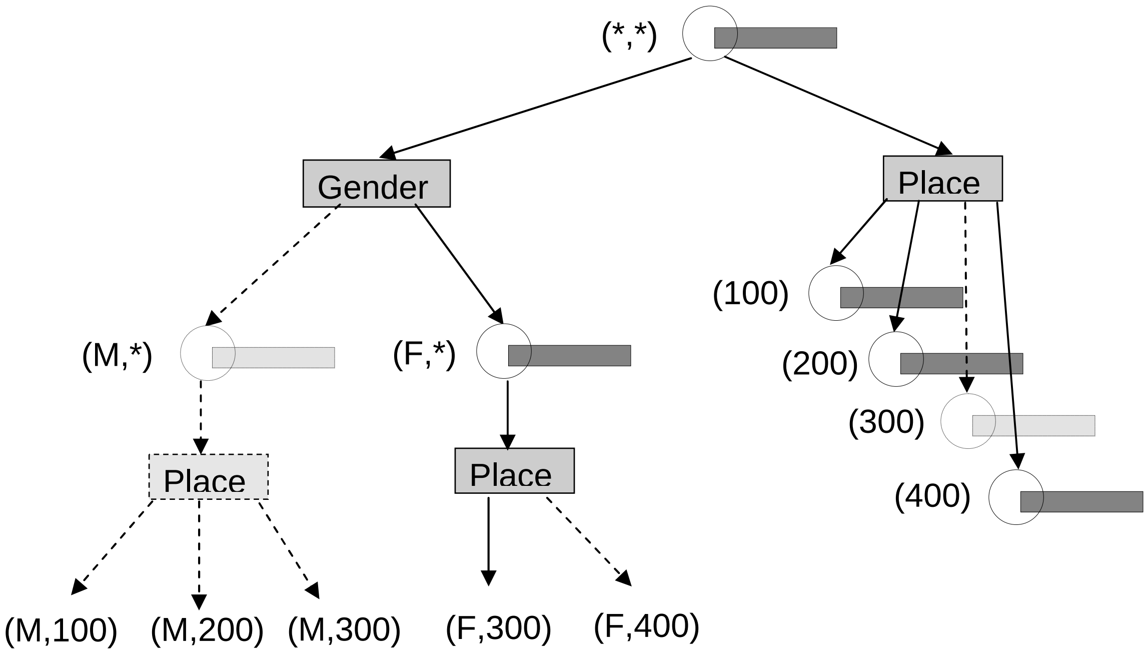

Figure 6. Illustration of the Most-Common-Value method.

Figure 6 illustrates this idea using an AD-Tree built for the sample dataset shown in Table 4. The nodes *(M,*), (F, 400), (300)* and everything below them can be removed from the tree. Any query that requires accessing the counts from the removed portions of the tree can be served by re-computing the missing numbers using the data stored in the remaining nodes. These re-calculations may require some navigation through the tree and a few basic arithmetic operations based on the fact that the sum of the counts corresponding to any sub-tree must be equal to the counts attributed to that sub-tree by its parent node. So, to retrieve the counts corresponding to the missing sub-tree one only needs to subtract the sum of the counts attributed to the existing nodes from the sum stored at the root. For instance, the count of records for a given day stored in the dataset shown in Table 4 which correspond to *Place = 300* can be restored as the total count of records on that day (which is stored in the root node of the tree) less the sum of the same day counts represented in the nodes corresponding to *Place = 100*, *Place = 200*, and *Place = 400*. Having figured out that number, one can also re-compute other removed entries, such as the count of male patients reporting from the area 300, *(M, 300)*, as the difference between the count for *Place = 300* and the count of female patients from that area, *(F, 300)*, which happens to be represented in the tree.

So indeed it is possible to remove one child from each *vary* node and not lose any information. It leads to substantial savings in memory requirements at modest increase in the average query response time. The extra time is required to re-compute the missing counts whenever necessary. The most-common-value (MCV) trick results in immense memory savings for datasets with demographic attributes having skewed distribution of their distinct values (such as SYN dataset). It may not be as beneficial when the values of demographic attributes are distributed more uniformly. For instance, if the MCV trick is applied to a *vary* node with 10,000 children each having an equally long leaf list, then we will have to add 10,000 time series at run time to respond to the MCV child query, which can be computationally expensive. In order to control such effects, we introduce a parameter called MCV fraction (γ) that is the ratio of the leaf list size of the MCV node and the leaf size of its parent vary node. The T-Cube algorithm will apply the MCV trick only if the observed ratio is higher than the MCV fraction threshold. This parameter helps to balance the memory savings against the average query response time. For $\gamma > 1.0$, the MCV trick will never be used. And for $\gamma = 0.0$, the MCV trick will be used always.

Figure 7 summarizes the observed memory savings for different values of γ. With the MCV trick, we can finally fit the T-Cube for SYN dataset into the main memory of the test machine (recall that its 29 binary attributes are 95% percent sparse and they have 0 as the MCV). It requires only about 100MB of memory as compared to more than 16GB it needed before. ED and OTC datasets also show savings in memory but not as significant as using the tree-depth method. This is because the values of demographic attributes are almost uniformly distributed in those data sets.

Figure 8 shows the effect of varying γ on the average simple query response time (depicted at a logarithmic scale in the graph). The response times against SYN dataset are the slowest (~50 milliseconds), because most of the randomly generated queries involve MCV values which require an extra time to be re-computed. For ED and OTC datasets, $\gamma = 0.4$ seems to be a reasonable value to pick when both memory and response time are considered.

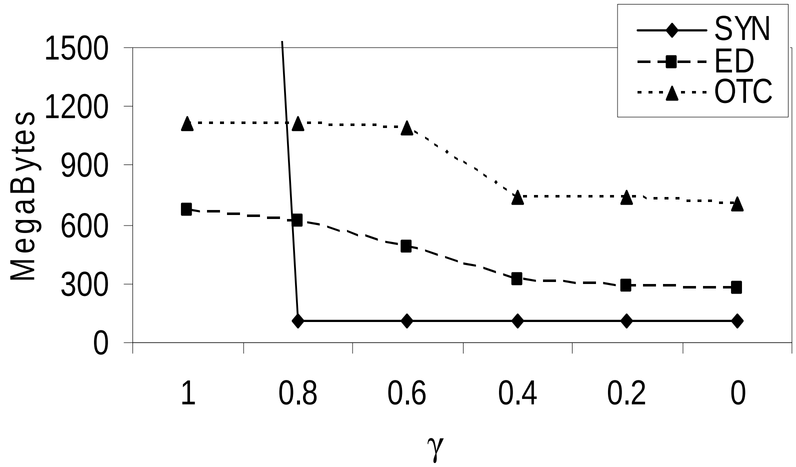

Figure 7. Memory savings attainable with the MCV method.

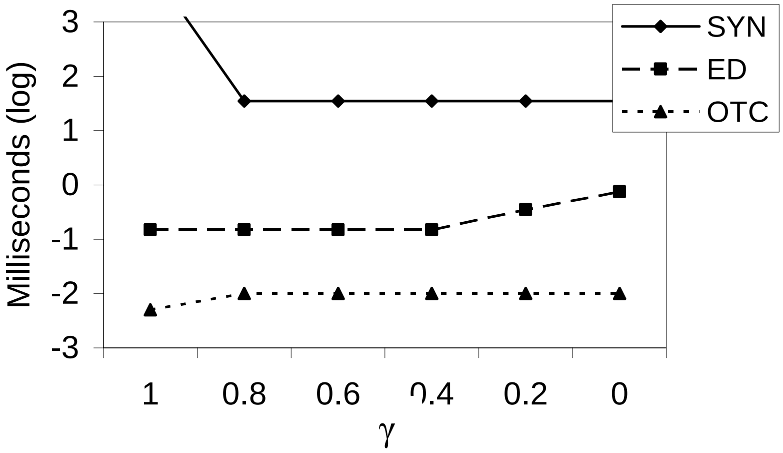

Figure 8. Average (simple) query response time when using the MCV method.

4.2.5 Responding to Complex Queries

T-Cubes can be built in minutes even for large data sets. Their memory requirements can be controlled and traded for response time in order to trim them to the manageable levels (of a few hundreds megabytes required to store AD-Trees for all of the considered three datasets). Also, the average response time to a simple query can be maintained within a fraction of a millisecond.

We define complex queries as those which are not strictly conjunctive, but allow each demographic attribute to take multiple values simultaneously. A query which requests counts for 1,000 *Zip Codes* is more complex than the one which only requests it for 10 of them. Let us define complexity of a query, β, as the fraction of possible unique values of an attribute which can be present in the complex query. Values of β belong to the range of [0,1], and the higher they are the higher complexity of a query.

Figure 9 shows the performance of T-Cube at different values of β (the results are averaged over 1,000 randomly picked queries). The parameters for the tested datasets were as follows: *r-value* = 1,000 and γ = 0.8. The observed average response time was within 100 milliseconds for SYN data and in an order of 1 millisecond for ED and OTC datasets. The results indicate that T-Cubes need only milliseconds to respond to even highly complex queries.

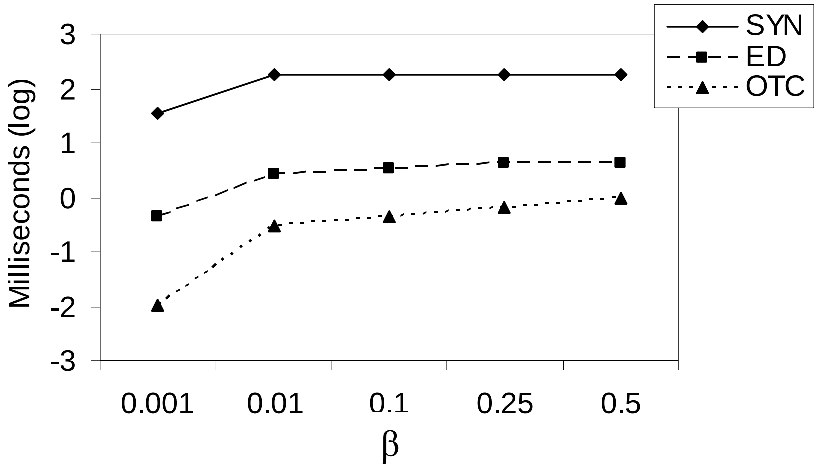

Figure 9. Average response time to complex queries.

4.2.6 Comparison with Commercial Tools

We also compared performance of T-Cube against other data cube tools available commercially. Due to privacy concerns we do not list the names of these tools here, but most of these tools are commonly used in many practical OLAP applications involving time series data. The data used for these tests had three demographic attributes with arities of 1,000, 10, and 5 respectively, 12 million records of daily transactions and covered a period of one year. The experiments were performed on a system with 2.4 GHz CPU ands 2 GB memory, running Windows XP operating system.

Table 5 shows the results of the comparison. Each of the commercial tools required a different amount of memory to represent the test data, however for all tools, the response time improved with the increase of the amount of used memory. Still though, the commercial tools require seconds to respond to a complex query. T-Cube on the other hand is able to respond in milliseconds, i.e. 1,000 times faster than the commercial tools. The two versions of T-Cubes (row 4 & 5) differ in the value of γ, the MCV fraction. The first one uses γ = 1,000 and in the second γ was set to 10 in order to illustrate the trade-off between memory consumption and response time attainable with the T-Cubes.

Table 5. Performance comparison: T-Cube vs. commercial tools.

Query Engine	Type	Memory	Response Time
Tool 1	RDMS	330 MB	6.8 sec
Tool 2	In memory	231 MB	7.6 sec
Tool 3	In memory	1+ GB	3.5 sec
T-Cube	In memory	236 MB	22 milliseconds
T-Cube	In memory	845 MB	5 milliseconds

4.2.7 Early Observations from Fielded Applications

The authors had a chance to see T-Cubes used in practice as enabling technology in applications requiring massive screening of multidimensional temporal data. These applications include systems to support monitoring of food and agriculture safety developed at the US Department of Agriculture and the Food and Drug Administration, as well as a system to monitor maintenance and supply data operated by the US Air Force.

One of these projects involved a data base consisting of datasets each with about 25 demographic attributes of arities varying from 2 to 80, about 12 thousand records of transactions, covering 6 years at a daily resolution. The application called for a massive screening through all combinations of attribute-value pairs of size = 1 and 2, the total number of such combinations exceeding 4.3 million. The involved analytics was based on an expectation-based temporal scan used to detect unusual short-term increases in counts of specific aggregate time series. The total number of individual temporal scan tests for one such data set exceeded 9.3 billion. Each such test involved a Chi-square test of independence performed on a 2-by-2 contingency table formed by the counts corresponding to the time series of interest (one of the 4.3 million series) and the baseline counts, within the current temporal window of interest (one of 2,000+) and outside of it. The complete set of computations, including the time necessary to retrieve and aggregate all the involved time series, compute and store the test results, load source data and build the T-Cube structure, etc., took about 8 hours of time when executed on a dual CPU AMD Opteron 242 1,600 MHz machine, in the 64 bit mode, using 1MB per CPU level 2 cache and 4 GB of main memory, running under Cent OS 4 Linux operating system. If the users used one of the commercial database tools, the time needed to retrieve a time series data corresponding to one of the involved queries would approach 180 milliseconds. Therefore, without the T-Cube, it would take about 9 days to just to pull all the required time series data from the database, not including any processing or executing statistical tests. That kind of analysis would be considered infeasible without the efficiencies provided by the T-Cube representation.

5 Conclusions

T-Cube is an efficient tool for representing additive time-series data labeled with a set of symbolic attributes. It is especially useful for retrieving responses to ad-hoc queries against large datasets of that kind. We showed that typically a T-Cube performs that task around 1,000 times faster than currently available commercial tools. This efficiency is very useful both in on-line investigation and in statistical data mining application scenarios. Rapid aggregation of time series across large sets of data made possible by T-Cubes becomes an enabling capability which makes manual lookups as well as many complex analyses feasible. T-Cube can be used as a general tool for any application requiring access to time series data from a database. From the application's perspective it is transparent: it acts just like the database itself, but an incredibly quickly responding one.

The size of memory used by T-Cube can be finely controlled by managing the tree-depth or using the MCV trick. There is trade-off between memory usage and the average query response time, and allowing more memory to be used by T-Cube leads to shorter response times. The experiments indicate that by selecting the right set of parameters one can achieve considerable memory savings at only linear increases in response time. The realistically-sized datasets we have tested so far were manageable in that their T-Cube representation would fit in less than 1GB of memory. This shows that T-Cubes can be used in practice to speed up access to large sets of time series data even on regular personal computers, alleviating the need for large and expensive servers.

T-Cubes are simple to setup and easy to use. Typically, it takes only minutes to build one from data. Database users do not need to define any stored procedures, or materialized views in order to make that happen. Once a T-Cube is built, it is ready to respond to any simple or complex query. The power of quickly retrieving any ad-hoc query makes T-Cubes potentially very useful in a range of applications, including a few specifically known to the authors. They have a potential to change the way users (people as well as analytic software systems) deal with the time series datasets.

Acknowledgements

This work was partially supported by the Centers of Disease Control (award number R01-PH000028), the National Science Foundation (award number IIS-0325581), the United States Department of Agriculture (award number 533A940311 via SAIC task order 4400134507), the United States Air Force (award number FA8650-05-C-7264) and the Food and Drug Administration (CIO-SP2i-TO-C-2426 via SAIC task order 4400120911). The authors would like to thank colleagues from the University of Pittsburgh for providing us the access to the bio-surveillance data sets (ED and OTC). We would also like to thank colleagues from the CMU Auton Lab: Karen Chen, Mike Baysek and Saswati Ray, for their help in conducting the experimental work:

References

[1] J. Han, M. Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, 2000.

[2] A. Moore, M. Lee. Cached sufficient statistics for efficient machine learning with large datasets. Journal of Artificial Intelligence research, 8:67-91, 1998.

[3] M. Tsui, J. Espino, M. Wagner. The Timeliness, Reliability, and Cost of Real-time and Batch Chief Complaint Data. RODS Laboratory Technical Report, 2005.

[4] M. Sabhnani, D. Neill, A. Moore, F. Tsui, M. Wagner, and J. Espino. Detecting anomalous patterns in pharmacy retail data. Proceedings of the KDD 2005 Workshop on Data Mining Methods for Anomaly Detection, 2005.

[5] B. Anderson, A. Moore. AD-trees for Fast Counting and for Fast Learning of Association Rules. Knowledge Discovery from Databases Conference, 1998.

[6] P. Komarek, A. Moore. A Dynamic Adaptation of AD-trees for Efficient Machine Learning on Large Data Sets. Proceedings of the 17th International Conference on Machine Learn-ing, 495-502, 2000.

[7] J. Gray, A. Bosworth, A. Layman, H. Pirahesh; Data Cube: A Relational Aggregation Op-erator Generalizing Group-By, Cross-Tab, and Sub-Total. ICDE 152-159, 1996.

[8] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: organization of and bulk in-cremental updates on the data cube. In Proceedings of the ACM SIGMOD Conference on Management of Data, pages 89-99, Arizona, May 1997.

[9] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In Proceedings of ACM SIGMOD '96, pages 205-216, Montreal, June 1996.

[10] M. Wagner, F. Tsui, J. Espino, V. Dato, D. Sittig, R. Caruana, L. McGinnis, D. Deerfield, M. Druzdzel, D. Fridsma. The emerging science of very early detection of disease outbreaks. Journal of Public Health Management Practice, Nov;7 (6):51-9, 2001.

ML

MACHINE LEARNING
DEPARTMENT

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon